Towards Scheduling Evolving Applications

Cristian Klein, Christian Pérez

Towards Scheduling Evolving Applications

Cristian KLEIN and Christian PÉREZ

INRIA/LIP, ENS de Lyon, France
cristian.klein@inria.fr, christian.perez@inria.fr

Abstract. Most high-performance computing resource managers only allow applications to request a static allocation of resources. However, evolving applications have resource requirements which change (evolve) during their execution. Currently, such applications are forced to make an allocation based on their peak resource requirements, which leads to an inefficient resource usage. This paper studies whether it makes sense for resource managers to support evolving applications. It focuses on scheduling fully-predictably evolving applications on homogeneous resources, for which it proposes several algorithms and evaluates them based on simulations. Results show that resource usage and application response time can be significantly improved with short scheduling times.

1 Introduction

High-Performance Computing (HPC) resources, such as clusters and supercomputers, are managed by a Resource Management System (RMS) which is responsible for multiplexing computing nodes among multiple users. Commonly, users get an exclusive access to nodes by requesting a static allocation of resources (i.e., a *rigid job* [1]), characterized by a node-count and a duration. Scheduling is mostly done using First-Come-First-Serve (FCFS) combined with backfilling rules such as EASY [2] or CBF [3]. Once the allocation has started, it cannot be grown nor shrunken.

As applications are becoming more complex, they exhibit **evolving** resource requirements, i.e., their resource requirements change during execution. For example, Adaptive Mesh Refinement (AMR) [4] simulations change the working set size as the mesh is refined/coarsened. Applications which feature both temporal and spatial compositions [5, 6] may have non-constant resource requirements as components are activated/deactivated during certain phases of the computation. Unfortunately, using only static allocations, evolving applications are forced to allocate resources based on their maximum requirements, which may lead to an inefficient resource utilisation.

We define three types of evolving applications. **Fully-predictably evolving** applications know their complete evolution at submittal. **Marginally-predictable** can predict changes in their resource requirements only some time in advance. **Non-predictably evolving** applications cannot predict their evolution at all.

This paper does an initial study to find out whether it is valuable for RMSs to support evolving applications. It focuses on fully-predictably evolving applications. While we agree that such an idealized case might be of limited practical use, it is still interesting to be studied for two reasons. First, it paves the way to supporting marginally-predictably evolving applications. If little gain can be made with fully-predictably evolving applications, where the system has complete information, it is clear that it makes little sense to support marginally-predictable ones. Second, the developed algorithms might be extensible to the marginally- and non-predictable case. Each time an application submits a change to the RMS, the scheduling algorithm for fully-predictable applications could be re-run with updated information.

The contribution of this paper is threefold. First, it presents a novel scheduling problem: dealing with evolving applications. Second, it proposes a solution based on a list scheduling algorithm. Third, it evaluates the algorithm and shows that significant gains can be made. Therefore, we argue that RMSs should be extended to take into account evolving resource requirements.

The remaining of this article is structured as follows. Section 2 presents related work. Section 3 gives a few definitions and notations used throughout the paper and formally introduces the problem. Section 4 proposes algorithms to solve the stated problem, which are evaluated using simulations in Section 5. Finally, Section 6 concludes this paper and opens up perspectives.

2 Related Work

Increased interest has been devoted to dynamically allocate resources to applications, as it has been shown to improve resource utilization [7]. If the RMS can change an allocation during run-time, the job is called **malleable**. How to write malleable applications [8, 9] and how to add RMS support for them [10, 11] has been extensively studied.

However, supporting evolving applications is different from malleability. In the latter case, it is the RMS that decides when an application has to grow/shrink, whereas in the former case, it is the application that requests more/fewer resources, due to some internal constraints.

The Moab Workload Manager supports so-called "dynamic" jobs [12]: the RMS regularly queries each application what its current load is, then decides how resources are allocated. This feature can be used to dynamically allocate resources to interactive workloads, but is not suitable for batch workloads. For example, let us assume that there are two evolving applications in the system, each using half of the platform. If, at one point, both of them require additional resources, a dead-lock occurs, as each application is waiting for the requested resources. Instead, the two applications should be launched one after the other.

In the context of Cloud computing, resources may be acquired on-the-fly. Unfortunately, this abstraction is insufficient for large-scale deployments, such as those required by HPC applications, because "out-of-capacity" errors may be encountered [13]. Thus, the applications' requirements cannot be guaranteed.

3 Problem Statement

To accurately define the problem studied in this paper, let us first introduce some mathematical definitions and notations.

3.1 Definitions and Notations

Let an **evolution profile (EP)** be a sequence of **steps**, each step being characterized by a *duration* and a *node-count*. Formally, $ep = \{(d_1, n_1), (d_2, n_2), \ldots, (d_N, n_N)\}$, where N is the number of steps, d_i is the duration and n_i is the node-count during Step i.

An evolution profile can be used to represent three distinct concepts. First, a **resource EP** represents the resource occupation of a system. For example, if 10 nodes are busy for 1200 s, afterwards 20 nodes are busy for 3600 s, then $ep_{res} = \{(1200, 10), (3600, 20)\}$.

Second, a **requested EP** represents application resource requests. For example, $ep_{req} = \{(500, 5), (3600, 10)\}$ models a two-step application with the first step having a duration of 500 s and requiring 5 nodes and the second step having a duration of 3600 s and requiring 10 nodes. Non-evolving, rigid applications can be represented by an EP with a single step.

Third, a **scheduled EP** represents the number of nodes actually allocated to an application. For example, an allocation of nodes to the previous two-step application might be $ep_s = \{(2000, 0), (515, 5), (3600, 10)\}$. The application would first have to wait 2000 s to start its first step, then it would have to wait another 15 s ($= 515\,\text{s} - 500\,\text{s}$) to start its second step.

We define the expanded and delayed EPs of $ep = \{(d_1, n_1), \ldots, (d_N, n_N)\}$ as follows: $ep' = \{(d'_1, n_1), \ldots, (d'_N, n_N)\}$ is an **expanded EP** of ep, if $\forall i \in \{1, \ldots, N\}, d'_i > d_i$; $ep'' = \{(d_0, 0), (d_1, n_1), \ldots, (d_N, n_N)\}$ is a **delayed EP** of ep, if $d_0 > 0$.

For manipulating EPs, we use the following helper functions:

- $ep(t)$ returns the number of nodes at time coordinate t,
  i.e., $ep(t) = n_1$ for $t \in [0, d_1)$, $ep(t) = n_2$ for $t \in [d_1, d_1 + d_2)$, etc.
- $\max(ep, t_0, t_1)$ returns the maximum number of nodes between t_0 and t_1,
  i.e., $\max(ep, t_0, t_1) = \max_{t \in [t_0, t_1)} ep(t)$, and 0 if $t_0 = t_1$.
- $\text{loc}(ep, t_0, t_1)$ returns the end-time of the last step containing the maximum, restricted to $[t_0, t_1]$,
  i.e., $\text{loc}(ep, t_0, t_1) = t \Rightarrow \max(ep, t_0, t) = \max(ep, t_0, t_1) > \max(ep, t, t_1)$.
- $\text{delay}(ep, t_0)$ returns an evolution profile that is delayed by t_0.
- $ep_1 + ep_2$ is the sum of the two EPs, i.e., $\forall t, (ep_1 + ep_2)(t) = ep_1(t) + ep_2(t)$.

3.2 An RMS for Fully-Predictably Evolving Applications

To give a better understanding on the core problem we are interested in, this section briefly describes how fully-predictably evolving applications could be scheduled in practice.

Let us consider that the platform consists of a homogeneous cluster of n_{nodes} computing nodes, managed by a centralized RMS. Fully-predictably evolving applications are submitted to the system. Each application i expresses its resource requirements by submitting a requested EP[1] $ep^{(i)}$ ($ep^{(i)}(t) \leq n_{nodes}, \forall t$). The RMS is responsible for deciding when and which nodes are allocated to applications, so that their evolving resource requirements are met.

During run-time, each application maintains a session with the RMS. If from one step to another the application increases its resource requirements, it keeps the currently allocated nodes and has to *wait* for the RMS to allocate additional nodes to it. Note that, the RMS can *delay* the allocation of additional nodes, i.e., it is allowed to expand a step of an application. However, we asssume that during the wait period the application cannot make any useful computations: the resources currently allocated to the application are **wasted**. Therefore, **the scheduled EP** (the EP representing the resources effectively allocated to the application) must be equal to the requested EP, optionally **expanded** and/or **delayed**.

If from one step to another the node-count decreases, the application has to release some nodes to the system (the application may choose which ones). The application is assumed fully-predictable, therefore, it is not allowed to contract nor expand any of its steps at its own initiative.

A practical solution to the above problem would have to deal with several related issues. An RMS-Application protocol would have to be developed. Protocol violations should be detected and handled, e.g., an application which does not release nodes when it is required to should be killed. However, these issues are outside the scope of this paper.

Instead, this paper does a preliminary study on whether it is meaningful to develop such a system. For simplicity, we are interested in an offline scheduling algorithm that operates on the queued applications and decides how nodes are allocated to them. It can easily be shown that such an algorithm does not need to operate on node IDs: if for each application, a scheduled EP is found, such that the sum of all scheduled EPs never exceeds available resources, a valid mapping can be computed at run-time. The next section formally defines the problem.

3.3 Formal Problem Statement

Based on the previous definitions and notations, the problem can be stated as follows. Let n_{nodes} be the number of nodes in a homogeneous cluster. n_{apps} applications having their requested EPs $ep^{(i)}$ ($i = 1 \dots n_{apps}$) queued in the system ($\forall i, \forall t, ep^{(i)}(t) \leq n_{nodes}$). The problem is to compute for each application i a scheduled EP $ep_s^{(i)}$, such that the following conditions are simultaneously met:

C1 $ep_s^{(i)}$ is equal to $ep^{(i)}$ or a delayed/expanded version of $ep^{(i)}$ (see above why);
C2 resources are not overflown ($\forall t, \sum_{i=1}^{n_{apps}} ep_s^{(i)}(t) \leq n_{nodes}$).

Application completion time and resource usage should be optimized.

[1] Note that this is in contrast to traditional parallel job scheduling, where resource requests only consist of a node-count and a wall-time duration.

4 Scheduling Fully-Predictably Evolving Applications

This section aims at solving the above problem in two stages. First, a list-scheduling algorithm is presented, which transforms requested EPs into scheduled EPs. It requires a `fit` function which operates on two EPs at a time. Second, several algorithms for computing a `fit` function are described.

4.1 An Algorithm for Offline Scheduling of Evolving Applications

Algorithm 1 is an offline scheduling algorithm that solves the stated problem. It starts by initializing ep_r, the resource EP, representing how resource occupation evolves over time, to the empty EP. Then, it considers each requested EP, potentially expanding and delaying it using a helper `fit` function. The resulting scheduled EP $ep_s^{(i)}$ is added to ep_r, effectively updating the resource occupation.

The `fit` function takes as input the number of nodes in the system n_{nodes}, a requested EP ep_{req} and a resource EP ep_{res} and returns a time coordinate t_s and ep_x an expanded version of ep_{req}, such that $\forall t, ep_{res}(t) + \text{delay}(ep_x, t_s)(t) \leq n_{nodes}$. A very simple `fit` implementation consists in delaying ep_{req} such that it starts after ep_{res}.

Throughout the whole algorithm, the condition $\forall t, ep_r(t) \leq n_{nodes}$ is guaranteed by the post-conditions of the `fit` function. Since at the end of the algorithm $ep_r = \sum_{i=1}^{n_{apps}} ep_s^{(i)}$, resources will not be overflown.

4.2 The `fit` Function

The core of the scheduling algorithm is the `fit` function, which expands a requested EP over a resource EP. It returns a scheduled EP, so that the sum of the resource EP and scheduled EP does not exceed available resources.

Because it can expand an EP, the `fit` function is an element of the efficiency of a schedule. On one hand, a step can be expanded so as to interleave applications, potentially reducing their response time. On the other hand, when a step is expanded, the application cannot perform useful computations, thus resources are wasted. Hence, there is a trade-off between the resource usage, the application's start time and its completion time.

In order to evaluate the impact of expansion, the proposed `fit` algorithm takes as parameter the **expand limit**. This parameter expresses how many times the duration of a step may be increased. For example, if the expand limit is 2, a step may not be expanded to more than twice its original duration. Having an expand limit of 1 means applications will not be expanded, while an infinite expand limit does not impose any limit on expansion.

Base `fit` Algorithm Algorithm 2 aims at efficiently computing the `fit` function, while allowing to choose different expand limits. It operates recursively for each step in ep_{req} as follows:

Algorithm 1: Offline scheduling algorithm for evolving applications.

Input: $ep^{(i)}, i = 1 \dots n_{apps}$, requested EP of the application i,
n_{nodes}, number of nodes in the system,
$\texttt{fit}(ep_{src}, ep_{dst}, n_{nodes}) \rightarrow (t_s, ep_s)$, a `fit` function
Output: $ep_s^{(i)}$, scheduled EP of application i

$ep_r \leftarrow$ empty EP ;
for $i = 1$ **to** n_{apps} **do**
  $t_s^{(i)}, ep_x^{(i)} \leftarrow \texttt{fit}(ep^{(i)}, ep_r, n_{nodes})$;
  $ep_s^{(i)} \leftarrow \text{delay}(ep_x^{(i)}, t_s^{(i)})$;
  $ep_r \leftarrow ep_r + ep_s^{(i)}$;

Algorithm 2: Base `fit` Algorithm

Input: $ep_{req} = \left\{ \left(d_{req}^{(1)}, n_{req}^{(1)}\right), \dots, \left(d_{req}^{(N_{req})}, n_{req}^{(N_{req})}\right) \right\}$, EP to expand,
$ep_{res} = \left\{ \left(d_{res}^{(1)}, n_{res}^{(1)}\right), \dots, \left(d_{res}^{(N_{res})}, n_{res}^{(N_{res})}\right) \right\}$, destination EP,
n_{nodes} : number of nodes in the system,
l : maximum allowed expansion ($l \geq 1$),
i : index of step from ep_{req} to start with (initially 1),
t_0 : first moment of time where ep_{req} is allowed to start (initially 0)
Output: ep_x : expanded ep_{req},
t_s : time when ep_x starts **or** time when expansion failed

if $i > N_{req}$ **then**
  $t_s \leftarrow t_0$; $ep_x \leftarrow$ empty EP ; **return**
$d \leftarrow d_{req}^{(i)}$; $n \leftarrow n_{req}^{(i)}$; `/* duration and node-count of current step */`
$t_s \leftarrow t_0$;
while *True* **do**
  if $n_{nodes} - \max(ep_{res}, t_s, t_s + d) < n$ **then**
    $t_s \leftarrow \text{loc}(ep_{res}, t_s, t_s + d)$; **continue**
  if $i > 1$ **then**
    $t_{eas} \leftarrow t_s - l \cdot d_{req}^{(i-1)}$ `/* earliest allowed start of previous step */`
    if $t_{eas} > t_0 - d_{req}^{(i-1)}$ **then**
      $t_s \leftarrow t_{eas}$; $ep_x \leftarrow \varnothing$; **return**
    else if $n_{nodes} - \max(ep_{res}, t_0, t_s) < n_{req}^{(i-1)}$ **then**
      $t_s \leftarrow \text{loc}(ep_{res}, t_0, t_s)$; $ep_x \leftarrow \varnothing$; **return**
  $t_s^{tail}, ep_x \leftarrow \texttt{fit}(ep_{req}, ep_{res}, n_{nodes}, i + 1, t_s + d)$;
  if $ep_x = \varnothing$ **then**
    $t_s \leftarrow t_s^{tail}$; **continue**
  if $i > 0$ **then** prepend $(t_s^{tail} - t_s, n)$ to ep_x ;
  else
    prepend (d, n) to ep_x ;
    $t_s \leftarrow t_s^{tail} - d$;
  return

1. find t_s, the earliest time coordinate when the current step can be placed, so that n_{nodes} is not exceeded (lines 4 – 7);
2. test if this placement forces an expansion on the previous step, which exceeds the expand limit (lines 8 – 11) or exceeds n_{nodes} (lines 12 – 13);
3. recursively try to place the next step in ep_{req}, starting at the completion time of the current step (line 14);
4. prepend the expanded version of the current step in ep_x (line 17). The first step is delayed (i.e., t_s is increased) instead of being expanded (line 20).

The recursion ends when all steps have been successfully placed (lines 1–2).

Placement of a step is first attempted at time coordinate t_0, which is 0 for the first step, or the value computed on line 14 for the other steps. After every failed operation (placement or expansion) the time coordinate t_s is increased so that the same failure does not repeat:

- if placement failed, jump to the time after the encountered maximum (line 7);
- if expansion failed due to the expand limit, jump to the first time which avoids excessive expansion (computed on line 11, used on line 16).
- if expansion failed due to insufficient resources, jump to the time after the encountered maximum (computed on line 13, used on line 16);

Since each step, except the first, is individually placed at the earliest possible time coordinate and the first step is placed so that the other steps are not delayed, the algorithm guarantees that the application has the earliest possible completion time. However, resource usage is not guaranteed to be optimal.

Post-processing Optimization (Compacting) In order to reduce resource waste, while maintaining the guarantee that the application completes as early as possible, a **compacting** post-processing phase can be applied. After a first solution is found by the base `fit` algorithm, the expanded EP goes through a compacting phase: the last step of the applications is placed so that it ends at the completion time found by the base algorithm. Then, the other steps are placed from right (last) to left (first), similarly to the base algorithm. In the worst case, no compacting occurs and the same EP is returned after the compacting phase.

The base `fit` algorithm with compacting first optimizes completion time then start time (it is optimal from expansion point-of-view), but because it acts in a greedy way, it might expand steps with high node-count, so it is not always optimal for resource waste.

4.3 Discussions

This section has presented a solution to the problem stated in Section 3.3. The presented strategies attempt to minimize both completion time and resource waste. However, these strategies treat applications in a pre-determined order and do not attempt to do a global optimization. This allows the algorithm to be easier to adapt to an online context in future work for two reasons. First, list scheduling algorithms are known to be fast, which is required in a scalable RMS implementation. Second, since the algorithms treat application in-order, starvation cannot occur.

5 Evaluation

This section evaluates the benefits and drawbacks of taking into account evolving resource requirements of applications. It is based on a series of experiments done with a home made simulator developed in Python. The experiments are first described, then the results are analyzed.

5.1 Description of Experiments

The experiments compare two kinds of scheduling algorithms: `rigid`, which does not take into account evolution, and variations of Algorithm 1. Applications are seen by the `rigid` algorithm as non-evolving: the requested node-cound is the maximum node-count of all steps and the duration is the sum of the durations of all steps. Then, `rigid` schedules the resulting jobs in a CBF-like manner.

Five versions of Algorithm 1 are considered to evaluate the impact of its options: base fit with no expansion (`noX`), base fit with expand limit of 2 without compacting (`2X`) and with compacting (`2X+c`), base fit with infinite expansion without compacting (`infX`) and with compacting (`infX+c`).

Two kinds of metrics are measured: system-centric and user-centric. The five system-centric metrics considered are: (1) *resource waste*, the resource area (nodes×duration, expressed as percent of total resources), which has been allocated to applications, but has not been used to make computations (see Section 3.2); (2) *resource utilisation*, the resource area that has been allocated to applications; (3) *effective resource utilisation*, the resource area (expressed as percent of total resources) that has been effectively used for computations; (4) *makespan*, the maximum of the completion times; (5) *schedule time*, the computation time taken by a scheduling algorithm to schedule one test on a laptop with an Intel®Core™2 Duo processor running at 2.53 GHz.

The five user-centric metrics considered are: (1) per-test average application completion time (`Avg. ACT`); (2) per-test average application waiting time (`Avg. AWT`); (3) the number of expanded applications (`num. expanded`) as a percentage of the total number of applications in a test; (4) by how much was an application expanded (`App Expansion`) as a percentage of its initial total duration; (5) per-application `waste` as a percentage of resources allocated to the application.

As we are not aware of any public archive of evolving application workloads, we created synthetic test-cases. A test case is made of a uniform random choice of the number of applications, their number of steps, as well as the duration and requested node-count of each step. We tried various combinations that gave similar results. Table 1 and 2 respectively present the results for the system- and user-centric metrics of an experiment made of 1000 tests. The number of applications per test is within $[15, 20]$, the number of steps within $[1, 10]$, a step duration within $[500, 3600]$ and the node-count per step within $[1, 75]$.

5.2 Analysis

Administrator's Perspective `rigid` is outperformed by all other strategies. They improve effective resource utilisation, reduce makespan and drastically reduce

Table 1. Comparison of Scheduling Algorithms (System-centric Metrics)

Name	Waste (%)			Utilisation (relative)			Eff. Util. (%)			Makespan (relative)			Sch. Time (ms)		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
`rigid`	43	70	116	1	1	1	30	40	51	1	1	1	4.64	6.2	9.41
`noX`	0	0	0	.46	.58	.69	49	61	73	.49	.65	.82	11.4	24.7	55.8
`2X`	0	2	11	.47	.60	.71	50	63	75	.48	.64	.82	11.4	24.4	45.4
`2X+c`	0	ϵ	4	.46	.59	.70	53	63	75	.48	.63	.82	17.1	36.7	88.6
`infX`	0	7	22	.49	.63	.78	52	64	73	.49	.63	.78	11.4	23.4	49.2
`infX+c`	0	1	11	.46	.59	.71	55	64	74	.47	.62	.78	17.6	36	124

Table 2. Comparison of Scheduling Algorithms (User-centric Metrics)

Name	Avg. ACT (relative)			Avg. AWT (relative)			Num. expanded (%)			App expansion (%)			Per-app. waste (%)		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
`rigid`	1	1	1	1	1	1	0	0	0	0	0	0	0	67	681
`noX`	.42	.61	.84	.36	.55	.81	0	0	0	0	0	0	0	0	0
`2X`	.45	.61	.84	.36	.54	.80	0	22	56	0	4	76	0	2	75
`2X+c`	.44	.60	.84	.37	.54	.81	0	7	40	0	ϵ	60	0	ϵ	41
`infX`	.43	.62	.81	.27	.53	.76	0	26	62	0	19	884	0	6	360
`infX+c`	.44	.60	.81	.35	.53	.76	0	13	47	0	5	1354	0	1	119

resource waste within reasonable scheduling time. Compared to `rigid`, all algorithms reduce resource utilization. We consider this to be a desired effect, as it means that, instead of allocating computing nodes to applications which do not effectively use them, these nodes are release to the system. The RMS could, for example, shut these nodes down to save energy.

There is a trade-off between resource waste and makespan (especially when looking at maximum values). However makespan differs less between algorithms than waste. If maintaining resources is expensive, an administrator may choose the `noX` algorithm, whereas to favour throughput, she would choose `2X+c`.

User's Perspective When compared to `rigid`, the proposed algorithms always improve both per-application resource waste and average completion time. When looking at maximum values, the trade-off between expansion / waste vs. completion time is again highlighted. Algorithms which favor stretching (`infX`, `infX+c`) reduce average waiting time, but not necessarily average completion time.

The results show that waste is not equally split among applications, instead, few applications are expanded a lot. Since most cluster / grid systems are subject to accounting (i.e., in a way, users pay for the resources that are allocated to them), using the `infX` and `infX+c` algorithm (which do not guarantee an upper bound on the waste) should be avoided. Regarding algorithms which limit expansion, the benefits of using `2X+c` instead of `noX` are small, at the expense of significant per-application resource waste. Therefore, users might prefer not to expand their applications at all.

Global Perspective From both perspectives, expanding applications has limited benefit. Therefore, the `noX` algorithm seems to be the best choice. Taking into account evolving requirements of applications enables improvement of all metrics compared to an algorithm that does not take evolvement into consideration.

6 Conclusions

Some applications, such as adaptive mesh refinement simulations, can exhibit evolving resource requirements. As it may be difficult to obtain accurate evolvement information, this paper studied whether this effort would be worthwhile in term of system and user perspectives. The paper has presented the problem of scheduling fully-predictable evolving applications, for which it has proposed an offline scheduling algorithm, with various options. Experiments show that taking into account resource requirement evolvement leads to improvements in all measured metrics, such as resource utilization and completion time. However, the considered expansion strategies do not appear valuable.

Future work can be divided into two directions. First, the algorithm has to be adapted to online scheduling. Second, as real applications are not fully-predictable, this assumption has to be changed and the resulting problem needs to be studied.

References

1. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling - a status report. In: JSSPP. (2004)
2. Lifka, D.: The ANL/IBM SP scheduling system. In: JSSPP. (1995) 295–303
3. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. TPDS **12**(6) (2001)
4. Plewa, T., Linde, T., Weirs, V.G., eds.: Adaptive Mesh Refinement – Theory and Applications. Springer (2003)
5. Bouziane, H., Pérez, C., Priol, T.: A software component model with spatial and temporal compositions for grid infrastructures. In: EuroPar. (2008)
6. Ribes, A., Caremoli, C.: Salome platform component model for numerical simulation. COMPSAC **2** (2007) 553–564
7. Hungershofer, J.: On the combined scheduling of malleable and rigid jobs. In: SBAC-PAD. (2004)
8. Buisson, J., Sonmez, O., Mohamed, H., et al.: Scheduling malleable applications in multicluster systems. Technical Report TR-0092, CoreGRID (2007)
9. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Dynamic malleability in iterative MPI applications. In: CCGRID. (2007)
10. C. Cera, M., Georgiou, Y., Richard, O., Maillard, N., O. A. Navaux, P.: Supporting MPI malleable applications upon the OAR resource manager. In: COLIBRI. (2009)
11. Buisson, J., Sonmez, O., Mohamed, H., et al.: Scheduling malleable applications in multicluster systems. Technical Report TR-0092, CoreGRID (2007)
12. Adaptive Computing Enterprises, Inc.: Moab workload manager administrator guide, version 6.0.2. http://www.adaptivecomputing.com/resources/docs/mwm.
13. Cycles, C.: Lessons learned building a 4096-core cloud HPC supercomputer. http://blog.cyclecomputing.com/2011/03/cyclecloud-4096-core-cluster.html.